# Thesis Title

by

Rouzbeh Paktinatkeleshteri

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

# Contents

0.1 Introduction . . . . . . . . . . 1
0.2 Background . . . . . . . . . . 2
0.3 Efficient Active-Lane Consolidation . . . . . . . . . . 6
0.3.1 Original ALC Design . . . . . . . . . . 6
0.3.2 How Gather/Scatter Instructions Hurt ALC . . . . . . . . . . 7
0.3.3 Efficient ALC via Data Permutation . . . . . . . . . . 9
0.3.4 Single Control-Flow-Dependent Path Case . . . . . . . . . . 10
0.4 ALC as a Compiler Transformation . . . . . . . . . . 11
0.4.1 ALC Analysis . . . . . . . . . . 11
0.4.2 ALC Transformation . . . . . . . . . . 13
0.4.3 Implementation in LLVM . . . . . . . . . . 14
0.5 Evaluation . . . . . . . . . . 14
0.5.1 Setup . . . . . . . . . . 14
0.5.2 Experimental Methodology . . . . . . . . . . 15
0.5.3 Data Permutation: More Instructions But Better Performance . . . . . . . . . . 16
0.5.4 Scatter Instructions Significantly Impact Performance . 17
0.5.5 Data Permutation Improves ALC . . . . . . . . . . 18
0.5.6 Conditionally Incremented Array Indexes . . . . . . . . . . 18
0.6 Related Work . . . . . . . . . . 19
0.7 Conclusion . . . . . . . . . . 21

**References** **23**

# List of Tables

1 Performance metrics when executing different versions of the loop in Listing 2: number of cycles to execute the loop (*Total Cycles*), number of executed instructions (*Num. Exec. Instructions*), cycles with no instruction completed (*Stalled Cycles*), and cycles stalled due to memory operations (*Memory Ops. Stalled Cycles*). Versions of the code: non-vectorized (`Scalar`), `if`-converted & vectorized (`if-conv`), Iteractive-ALC with gather/scatter instructions (`ALC`), and Iteractive-ALC with data permutation and without gather instructions (`ALC+DP`). . . . . . . . . . . . 8

# List of Figures

1 (a) Original CFG of Listing 1; (b) CFG after control-flow linearization (CFL); (c) CFG in (b) after inserting of `all_true` BOSCC. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

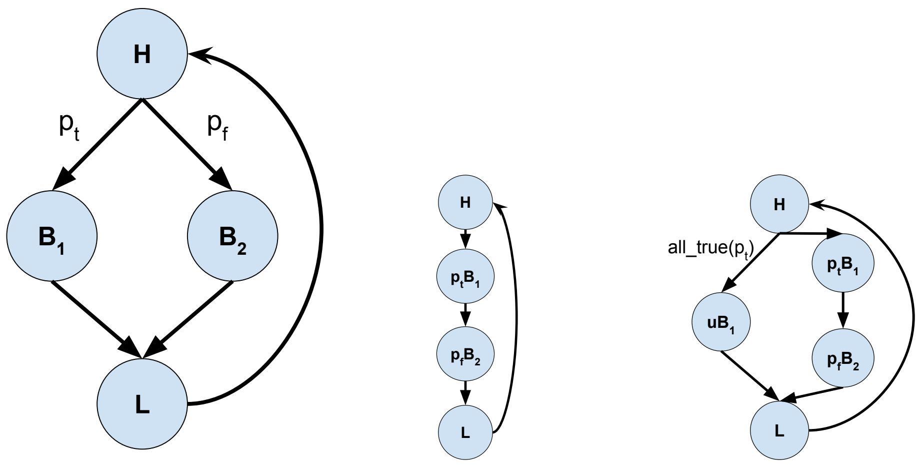
## 0.1 Introduction

Compilers have been successful in performing auto-vectorization for exploiting data-parallelism using Single-Instruction Multiple-Data (SIMD) instructions for decades [9], [11], [15], [18]. However, control-flow divergence in loops, found in scientific and high-performance applications, can hinder or stop compilers from generating SIMD instructions [2], [13]. Compilers address this issue by using predicated instructions that have an extra operand, a predicate register or predicate, that holds the value of the condition that needs to be true for the instruction to commit [2], [7], [13], [16], [17]. When a predicate is false, the corresponding data element is inactive with respect to predicate instructions. Vector predicates are bit vectors where each lane of the predicate vectors indicates if the corresponding lane in the associated data vector registers is active. An entire basic block may be guarded by the same predicate register. Although control-flow linearization (CFL) enables vectorization, the linearized code executes computations on vector registers with inactive lanes, wasting computational resources. Some techniques that address CFL limitations in vectorized code require the predicate vector to be dynamically uniform: all lanes for a given vector operation must be either active or inactive [10], [12]. Dynamic uniformity is less likely to occur in programs executed in processors with modern vector extensions, such as Intel's AVX512 and Arm's SVE that have longer vectors. Wyatt *et al.* propose Active-lane Consolidation (ALC) to address these issues [14]. The idea of ALC is to form uniform vectors dynamically by merging active lanes from different iterations. However, the seminal presentation of ALC only evaluated it in a simulation environment using hand-modified programs. This work presents the first evaluation of ALC on hardware — this evaluation is performed in the Fujitsu A64FX processor, the first processor to implement ARMv8.2-A SVE instruction — and the first automated code generation for ALC in the LLVM open-source compiler. This evaluation reveals a mismatch between the original ALC design's assumptions about the latency of gather instructions and the actual latency of these instructions in the A64FX. Thus, the paper also presents a re-design of ALC that leads to performance improvements over LLVM's existing vectorization technique for some cases. As such, the main contributions are:

- The first in-depth performance analysis of different implementations of ALC executing in a processor that implements SVE (Section ) that reveals limitations in the original ALC design (Section 0.3.2);
- A design improvement to ALC that uses data-permutation instructions instead of gather instructions and leads to the generation of code that is up to 4× faster than the original design (Section 0.3.3);
- The first automated generation of ALC code via a compiler transformation that generates code that is up to 79% faster than `if`-converted code produced by Arm's Clang, a production-ready compiler (Section 0.4);
- An ALC code-generation algorithm, specialized for the case where there is only a single control-flow-divergent path in the target loop, that combines the best of ALC and if-conversion (Section 0.3.4 ); and
- A discussion of remaining challenges in the path toward applying ALC to broader loop patterns (SectionSection 0.5.6).

## 0.2 Background

**Single-Instruction Multiple-Data** (SIMD) parallelism is available on modern processors through vector units. The vector instructions executed in these units encode operations to be performed on vector registers. Each data item in a vector register occupies a *vector lane*, or *lane*. The **length of vector** (VL) is the number of bits in a vector register, while the number of data items in a vector register is the **vector factor** (VF). Traditionally, VL was a constant known at compilation time, however novel **Instruction-Set Architecture** (ISAs) have vector-length agnostic vector instructions where the VL is not known at compilation time — and can even be changed at runtime by the hypervisor. An example of such a design is the Arm **Scalable Vector Exten-**



(a) Original. (b) Linearized. (c) BOSCC.

Figure 1: (a) Original CFG of Listing 1; (b) CFG after control-flow linearization (CFL); (c) CFG in (b) after inserting of `all_true` BOSCC.

**sions** (SVE) available on ARM v8.3 & v9 processors and Fujitsu's A64FX[1]. Vector units may also have *predicate vectors*, which are bit-vector registers where each bit indicates if a corresponding lane of the vector register is *active*. Instructions that accept predicate registers are known as *predicated instructions*, and they only operate on active lanes — inactive lanes are left unchanged. Predicated instructions can be used to enable the vectorization of loops that contain control-flow instructions.

```
for (i = 0; i < n; i++) {
    if (a[i] < b[i]) {
        a[i] = b[i] * c[i];
    } else{
        b[i] = a[i] + c[i];
    }
}
```

Listing 1: A control-flow divergent loop.

In a loop with *divergent* control flow, different iterations of the loop may execute instructions from different paths in the loop's **Control-Flow Graph** (CFG). Divergent control flow may be an obstacle to the vectorization of a loop. Common constructs in computer code, such as `if-then-else` and

[1]At the time of writing, A64FX is the only available processor that supports SVE.

`switch-case` statements may introduce divergence. Modern compilers are able to vectorize some control-flow divergent loops after applying a **Control-Flow Linearization** (CFL) technique known as `If`-Conversion. If conversion transforms control-flow dependencies into data dependencies [2], [13]. For instance, consider the `for`-loop in Listing 1 and its CFG in Figure 1a. The statements on Line 3 and Line 5 are control-flow dependent on the condition in Line 2. CFL eliminates control flow by first computing a predicate register for each possible path. Then the instructions on each basic block are predicated with the predicate registers that correspond to the condition that needs to be true for that block to be executed. For example, instructions in block $B_1$ are predicated with a predicate register $p_t$, where $p_t = (a[i] < b[i])$. $B_2$ are predicated with a predicate register $p_f$, where $p_f = (a[i] \leq b[i]) = \neg p_t$. Figure 1b shows the CFG in Figure 1a after CFL, where predicated blocks are denoted by prefixing the original block name with their corresponding predicate register.

Once the body of a loop is linearized via CFL, vectorization proceeds with the conventional recipe followed by most modern compilers, which consists in widening scalar operands into vector operands and replacing scalar operations with their equivalent vector versions. Scalar predicate registers are replaced with vector predicate registers. Predicated execution, or simply *predication*, of scalar instructions is supported on modern processors via predicated instructions. For vector instructions, predication is supported via predicated vector instructions or bit-wise vector instructions, in a process known as *masking*.

Although CFL enables the use of SIMD instructions, it clearly wastes resources and computational cycles because all instructions from each mutually exclusive path will be executed but only one path will have their computations committed. Generating **Branch-On-Superword-Condition-Codes** (BOSCCs) is a common technique to avoid executing instructions from paths where vector lanes would be inactive [7], [16], [17]. BOSCCs are instructions, or a sequence of instructions, that dynamically checks the uniformity of predicate registers. In a uniform predicate register the condition evaluates to the same value — all true or all false — for all lanes. In such cases, only the instructions corresponding *uniform path*, true path or false path, need to be executed. For example, after vectorization, in Figure 1b if $p_t$ is a uniform true vector, then only instructions in $B_1$ need to be executed and instructions in $B_2$ can be skipped. Figure 1c shows the CFG in Figure 1b after BOSCCs are inserted by the compiler. As Figure 1c shows, an `all_true` *guard condition* is generated to check if $p_t$ is a uniform true vector. In such a case, the control flow is directed to a uniform block $uB_1$, that only contains instructions from block $B_1$. When $p_t$ is a uniform true vector, the instructions in block $B_2$ can be skipped because $p_f = \neg p_t$, and thus all lanes would be inactive.

The insertion of BOSCCs can improve SIMD utilization and reduce the number of wasted cycles by avoiding executing vector instructions with all lanes inactive. However, the benefits of BOSCCs can only be observed if uniform vectors occur frequently. If uniformity is rare, then the use of BOSCCs does not increase SIMD utilization [14]. Moreover, the likelihood of uniformity decreases with increased VL, thus uniform predicate vectors are less likely to be found for architecture with long vectors — e.g. AVX512, Fujitsu A64Fx. **Active-Lane Consolidation** (ACL) is an algorithm proposed to increase SIMD utilization even in the presence of infrequent uniform vectors and architectures with long vectors [14]. At the core, ACL is a permutation algorithm that creates uniform vectors by merging active lanes from two, or more, non-uniform vectors into a *merged vector*. Permutation enables ALC to only execute non-predicated blocks with the constructed uniform vector and effectively avoid executing linearized code.

Praharenka *et al.* propose ALC as an algorithm and manually apply it to each evaluated benchmark. This work shows presents the first compiler-only optimization pass that automatically applies ALC to a loop that contains divergent control flow. Furthermore, during the seminal work of Praharenka *et al.* they had no access to a processor that implements SVE. Therefore, all their experimental results are based on simulations conducted with the Arm's Instruction Emulator (ArmIE). Therefore, Praharenka *et al.*'s work only shows improvements in terms of the reduction in the number of executed instructions. Our performance study on a hardware implementation of SVE revealed that some of the estimates for the latency of instructions used by Praharenka *et al.* were significantly off. This work shows that accounting for the actual instruction latencies in a hardware implementation of ALC requires a redesign of aspects of ALC To the best of our knowledge, this work is the first to show ALC's performance on real hardware. Moreover, it identified limitations on the original algorithm that will be discussed in the following section.

## 0.3 Efficient Active-Lane Consolidation

This section presents an ALC design that addresses key limitations in Wyatt *et al.*'s initial design[14]. After a review of the original ALC design (Section 0.3.1), Section 0.3.2 presents evidence that the higher-than-anticipated cost of the gather/scatter instructions renders Wyatt *et al.*'s ALC ineffective. The approach proposed in this work to eliminate gather/scatter instructions is presented in Section 0.3.3. Lastly, Section 0.3.4 describes a novel algorithm that extracts the best of both ALC and control-flow linearization in a common case when loops have only a single **control-flow-divergent path** (CFDP).

### 0.3.1 Original ALC Design

Wyatt *et al.* propose two variations of ALC: Unroll-ALC and Iteractive-ALC. In both versions, ALC is applied after `if`-conversion. In the Unroll-ALC, the `if`-converted and vectorized loop is unrolled once and two index vectors are formed, one for each iteration of the loop before unrolling. Each index vector is initialized such that each lane contains the value of the loop's induction variable of each scalar iteration.

In Wyatt *et al.*'s the path that is most likely to be taken across all iterations of the loop is chosen for consolidation based on profiling information. Then two predicate vectors are formed by evaluating the condition to execute that path. After doing permutation, all vector operations in the consolidated path operate with uniform vectors without predication. This work focuses on cases with small ($< 3$) paths and thus consolidation is applied to all paths.

Once the predicates are formed, the index vectors are permuted such that all active lanes are consolidated into a *merged vector* `vM`. The inactive lanes are kept in a *remainder vector* `vR`. Finally, if the `vM` is uniform, then the consolidated uniform path is executed. Otherwise, the `if`-converted path is executed. The main difference between Unroll-ALC and Iteractive-ALC is that, in Iteractive-ALC, the `if`-converted and vectorized loop is not unrolled. Instead, the active lanes from multiple iterations are merged into `vM` until `vM` is a uniform vector. Only then the consolidated uniform path is executed. Iteractive-ALC works well for loops with a conditional that contains vectorizable code in both the then and the else block. In this case, `then` lane are consolidated into `vM`, and `else` lane are consolidated into `vR`. Whenever either `vM` or `vR` is full, the corresponding code can be executed in a uniform vector.

In both versions, any data that is dependent on the loop indices — e.g. arrays $a$, $b$, and $c$ in Listing 1 —, are loaded using gather-load instructions because a consolidated vector might contain non-consecutive indices. A gather-load instruction loads data from (potentially) non-consecutive addresses calculated by adding a base-pointer operand to each index in the index-vector operand. Similarly, any write-back to memory needs to be performed via scatter-store instructions, which also can write into non-consecutive memory addresses. By design, the use of gather/scatter instructions is unavoidable in Wyatt *et al.*'s ALC. In Section 0.3.2, experimental results show that gather/scatter instructions can render ALC ineffective on real hardware with SVE.

### 0.3.2 How Gather/Scatter Instructions Hurt ALC

```
for (int i = 0; i < n; i++) {
    if (cond[i]) {
        b[i] = a[i];
    }
}
```

Listing 2: Simple conditional copy loop.

In order to understand the prohibitive overhead of gather/scatter instructions in ALC's performance, consider the simple loop in Listing 2. The loop conditionally copies elements from array $a$ to array $b$, both are 32-bit integers.

Table 1: Performance metrics when executing different versions of the loop in Listing 2: number of cycles to execute the loop (*Total Cycles*), number of executed instructions (*Num. Exec. Instructions*), cycles with no instruction completed (*Stalled Cycles*), and cycles stalled due to memory operations (*Memory Ops. Stalled Cycles*). Versions of the code: non-vectorized (`Scalar`), `if`-converted & vectorized (`if-conv`), Iteractive-ALC with gather/scatter instructions (`ALC`), and Iteractive-ALC with data permutation and without gather instructions (`ALC+DP`).

| Version / Metric | Loop Cycles | Num. Exec. Instructions | Stalled Cycles | Memory Ops. Stalled Cycles |
|---|---|---|---|---|
| `Scalar` | 132M | 224M | 21M | 1.6M |
| `if-conv` | 14M | 14M | 9M | 1.2M |
| `ALC` | 220M | 16M | 210M | 110M |
| `ALC+DP` | 58M | 63M | 38M | 1.3M |

An element $a[i]$ is copied to $b[i]$ if, and only if, the value in $cond[i]$ is true. Table 1 shows performance metrics for different versions of the loop in Listing 2. For the results in Table 1, the `cond` array was initialized such that every other element has a `true` value (50% sparsity). The results were obtained following the methodology in Section 0.5.2. Both `ALC` and `ALC+DP` versions are generated by the compiler pass described in Section 0.4.

Unsurprisingly, all vectorized versions of the loop — `if-conv`, `ALC`, `ALC`— execute fewer instructions than the `Scalar` code. Each vector instruction in the loop operates on 16 32-bit integers at a time ($VL = 512$ bits). However, `ALC` is more than 66% slower than `Scalar`, even while it executes 14× fewer instructions, because `ALC` causes 10× more stalls than the `Scalar` code. More than half of the stalls are due to waiting for memory stalls. The main culprits are the gather/scatter instructions because they require multiple load/store ports instead of a single port as regular vector loads [3]. In addition, in the current ARM vector-unit design, the address calculations for gather/scatter instructions are executed in the floating-point vector units [3], which have higher latency than the integer operations used for regular vector loads/stores. When Wyatt *et al.* evaluated their design, they had no access to hardware and thus based their evaluation on counting the number of instructions executed in a simulator. Table 1 are the first results, obtained in real hardware with SVE, that show that the reduction in the number of instructions enabled by Wyatt *et al.*'s ALC design does not translate into faster execution. Therefore, in order for an `ALC` design to have a chance at being faster than `if`-converted and vectorized code, gather/scatter instructions need to be avoided or eliminated. Section 0.3.3 discusses how gather instructions can be eliminated. In Section 0.5.4 shows empirical results which indicate that reducing the number of scatter stores also improves ALC's performance.

### 0.3.3 Efficient ALC via Data Permutation

Permuting both the indices and data vectors eliminates gather instructions. Figure **??** contrasts the differences in the code generated by the compiler from the example code in Listing 1. In Figure **??** the data for both consolidated paths are loaded with gather instructions and the permuted index vector `vM`. In contrast, `ALC+DP` eliminates gather instructions by also permuting the data vectors (`v1`, `v2`, and `v3`), as Lines **??**-**??** in Figure **??** shows. As a result, `ALC+DP` benefits from the same spatial locality and data prefetching as the `if`-converted code (Figure **??**). When `vM` is not uniformly true, then it is guaranteed that `vR` is uniformly false because there are only two CFDPs. This observation allows the compiler to generate an optimized version of Iteractive-ALC where loads for the first iteration of the loop are peeled. In Figure **??** and Figure **??** `vM` is initialized with the index vector of from the peeled iteration. Similarly, vectors `v1M`, `v2M`, and `v3M` are initialized with the data loaded from $a$, $b$, and $c$ as in the first iteration of the `if`-converted & vectorized loop (Figure **??**). With the above optimization, most loop iterations operate with fully uniform vectors leading to better utilization of the SIMD units.

Table 1 shows that eliminating gather instructions via data permutation (`ALC+DP`) significantly improves the performance of ALC. In particular, `ALC+DP` has over 5× less stalled cycles and executes in 3.8× fewer cycles than `ALC`, even though it executes almost 4× more instructions. This result indicated that reducing the number of executed instructions does not necessarily translate into better performance. Moreover, `ALC+DP` has over $84\times$ fewer stalls due to waiting for memory operations, as Table 1 shows. Memory stalls are significantly reduced because data vectors are loaded from consecutive memory locations, which benefit from the higher spatial locality and more accurate prefetching. On the other hand, gather instructions suffer from higher latencies in the address calculation and poor spatial locality of data elements. Therefore, the results in Table 1 indicate that trading off the execution of more instructions with avoiding gather instructions pays off. Data permutation adds vector-vector instructions, which have significantly lower latency than sophisticated memory instructions, such as gather/scatter instructions [3].

Table 1 shows that `ALC+DP` does not perform better than `if-conv` for the code in Listing 2. There is little room for ALC to save cycles by not executing vector instructions with inactive lanes because the simple loop does not perform enough work. ALC can only outperform `if`-conversion & vectorization on loops that have a sufficient number of instructions to hide the permutation overhead. In addition, a more significant number of instructions on CFDPs translates to more saved cycles, and fewer executed instructions, for loops with mutually exclusive paths.

### 0.3.4 Single Control-Flow-Dependent Path Case

Loops with a single CFDP, as the example in Listing 2, are a special case where the ALC design can be modified to extract the best of both `if`-conversion and ALC. In such loops, vector lanes are wasted with inactive elements, but the instructions on the single CFDP are the only source of wasted operations. Therefore, `if`-conversion & vectorization is a good solution when the majority of lanes in the predicate vector are active, but in the complementary case — loops with very sparse predicate vectors — `if`-converted & vectorized code executes a significant number of instructions that waste vector lanes.

With this in mind, the ALC algorithm needs to be modified such that it benefits from both permutation and `if`-conversion. Prior to index and data vector permutation, in each iteration, the number of active lanes can be calculated via population-count instructions that are available on most modern

ISAs (e.g. ARMv8.2-A & v9). If there are more active elements than the *vector factor* (VF) on *both* index vector, then the `if`-converted code is executed resulting in few wasted lanes. Otherwise — when there are fewer active lanes than VF — an ALC path is executed where vectors are permuted and the consolidated-uniform path is executed when the predicate vector is uniform. `vR` will be fully inactive because the vector permutation only happens when fewer than VF lanes are active. Therefore, there is no need for the compiler to emit both instructions to produce `vR` and the vector register itself. This observation allows the reduction in instruction overhead of the permutation logic by 50%.

## 0.4 ALC as a Compiler Transformation

Like most optimizations in modern compilers, the ALC transformation is designed with two components: (i) an analysis that identifies candidates loops to apply ALC based on loop features, such as the number of instructions on each CFDP and the CFG complexity (Section 0.4.1); and (ii) a transformation that applies active-lane consolidation to candidate loops that have enough instructions in each CFDP and low memory-to-compute instruction ratio to amortize vector-permutation costs (Section 0.4.2).

### 0.4.1 ALC Analysis

The main goals of the analysis are two-fold: 1. to identify candidate loops to apply ALC by checking for the legality of applying ALC; and 2. decide if a given candidate would benefit from the ALC transformation. It is legal to apply ALC to a loop $L$ with control-flow divergence if $L$ does not have *loop-carried dependencies* and only contains calls to vectorizable functions without side effects (e.g. square-root and sine & cosine). Similar to other forms of vectorization such as loop vectorization and SLP vectorization, ALC cannot be applied to loops with loop-carried dependencies because instructions from different iterations that depend on each other, usually, cannot execute in parallel in a SIMD fashion. The ALC analysis relies on existing data-dependency analysis available on modern compilers to identify loops without loop-carried dependencies. A data-dependency analysis may return a *may depend* answer because of unresolved alias relationships and, in that case, the compiler must conservatively not apply the transformation [5], [8].

The cost portion of the profitability analysis for ALC uses the estimated cost of executing the new instructions required to perform index and data-vector permutations for ALC minus the instructions that are eliminated by ALC. The estimated execution latency of instructions is generally available in a modern compiler because the same information is used in other transformations, such as the creation of an efficient instruction schedule. The benefit of ALC results from the increased utilization of SIMD units due to the consolidation of loop iterations with the same predicate on the same vector. Thus the benefit depends on the number of instructions and on the latency of the instructions executed in each of the control-flow paths in the loop. The distribution of true and false predicates in the iterations of the loop also affects the benefit of ALC. However, this information can usually only be obtained from profiling, thus it is not used in the profitability analysis for static compilation.

Empirically, and as the results in Section 0.5 support, the following factors are key when making the decision of whether or not to apply ALC: 1. Number of instructions on each CFDP; 2. Number of store operations in the loop (See Section 0.5.4); 3. Complexity of Loop's CFG (See Section 0.5.5). ALC should be applied to loops that have enough instructions in each CFDP so that the overhead of index and data-vector permutation can be amortized. As the results in Section 0.3.2 indicate, gather/scatter instructions can significantly hurt ALC's performance. Although gather instructions can be eliminated through data permutation, it is not possible to avoid all scatter instructions. Therefore, a loop that has a high ratio of memory-access instructions to compute instructions is not likely to benefit from ALC. In the current prototype, a conservative approach is used due to the absence of accurate branch probability information: ALC is only applied to loops with two, or fewer, CFDPs. Future development may seek to integrate branch probability information in the profitability analysis of ALC or may explore a more aggressive application of

ALC even in the absence of such information. If the compiler does not know which path is more likely to be taken, then active-lane consolidation needs to be applied for each CFDP. Such unbounded application of ALC is not likely to achieve better performance than `if`-converted code because of the permutation costs for all paths. Except for the branch probabilities, all other information can be obtained statically by a compiler, avoiding any need for profiling, in contrast to Wyatt *et al.*'s work [14].

In some loops with more than two CFDPs, it may be legal to apply loop fission to generate multiple loops in such a way that at least some of the resulting loops have less than two CFDPs. The condition to apply loop fission to a loop is that no loop-independent dependency may cross the point where the loop is split. Profitability analysis for loop fission has to take into account that after fission some loops may exhibit worse cache locality, for instance when different paths use/load the same data. The study of combining loop fission with ALC is left for future work.

### 0.4.2 ALC Transformation

The actual loop transformation to perform ALC is quite straightforward. First, scalar instructions that compute the conditions that control each CFDP are replaced with their vector equivalents, forming predicate vectors. After that, if the loop being transformed has only a single CFDP, then the transformation produces the code that chooses between the `if`-converted path and ALC path, as discussed in Section 0.3.4. For loops with two CFDPs, the code produced chooses between one of the two consolidated uniform paths (see Section 0.3.3). Then, for each CFDP, a consolidated uniform path is generated by traversing each scalar instruction in the path and generating its vector equivalent. After that, and only if the loop has a single CFDP, the transformation generates the `if`-converted path. For loops with a number of iterations that is not a multiple of VF, the transformation generates a remainder scalar loop to process the remaining elements. Loops with fewer than VF elements are left untouched by the ALC transformation.

### 0.4.3 Implementation in LLVM

A prototype implementation developed in LLVM is used to evaluate both ALC analysis & transformation passes in this work. LLVM is a *de-facto* standard compiler framework widely used by both industry and academia. The ALC analysis reuses the memory dependency and alias analysis used by LLVM's loop vectorized pass to determine if loops are loop-carried dependency free. Both analysis & transformation are passes in LLVM's intermediate representation (IR) level. For now, the transformation pass only generates code for Arm's SVE architecture, and it accomplishes this by using SVE intrinsics available at IR level in LLVM. However, all architecture-specific intrinsic calls are generated through an interface that, with minimum effort, can be extended to generate code for other architectures that also support SVE.

## 0.5 Evaluation

This section evaluates ALC as a compiler transformation by applying data permutation to a set of control-divergent loops and comparing its effectiveness to previous implementations of ALC and to code generated by existing production compilers. This evaluation aims to answer the following questions:
① Can Data Permutation improve ALC performance by reducing memory stalls?
② Do scatter instructions impact ALC's performance?
③ Should ALC be applied on loops with a single CFDP?

The results indicate that data permutation leads to a significant speedup of up to 79% over `if`-converted code and outperforms state-of-the-art compiler-based approaches.

### 0.5.1 Setup

The experiments run on a machine equipped with a Fujitsu A64FX locked at 1.8GHz that has access to 32GiB of RAM and runs Rocky Linux (release 8.4). A64FX is the first processor that implements ARMv8.2-A SVE instruction and it operates with 512-bit vector registers. The evaluation micro-benchmarks were developed in-house and are written in the C language. Each micro-benchmark is designed to be representative of application code and to ease the identification and measurement of factors that impact the performance of both `if`-converted code and ALC-transformed code. Wyatt *et al*. seminal work predicted the performance of ALC using hand-modified versions of applications in the SPEC CPU2017 benchmark suite running on a functional simulator [1]. Those modifications included removing control-flow dependent paths that execute I/O operations — a transformation that cannot be safely implemented in a compiler because it alters the behavior of the program. This work aims to evaluate the automated generation of ALC code by a compiler transformation pass and thus can only be applied to programs to which the compiler can safely apply ALC without user intervention. The micro-benchmarks used in this evaluation are comprehensive and share characteristics with loops found in the SPEC CPU2017 benchmark Suite. The limitations and challenges of the current ALC implementation are discussed in Section 0.5.6.

The evaluation micro-benchmarks consist of loops that contain either an `if-then-else` or a single `if` statement. Two micro-benchmarks help understand the impact of the factors discussed in Section 0.4.1: 1. `if-then-else` contains a loop that executes $N$ times with two CFDPs in its body. Each CFDP has: 20 arithmetic, 2 load, and 3 store instructions; 2. `if-then` also contains a loop that executes $N$ times but with a single CFDP, which has: 20 arithmetic, 2 load, and 3 store instructions. $N$ is set to five million for all experiments. Complete source code of ALC's LLVM IR pass and evaluation micro-benchmarks is available as part of this paper's **ARTIFACT**[2]. All source code is compiled with Clang/LLVM version 15.0.0[3] and highest optimization level enabled (`-O3`).

## 0.5.2 Experimental Methodology

Results presented in the following sections are the average of one hundred executions of each program. Very little variation was observed between mea-

[2]**link omitted due to double-blind reviews.**

[3]61baf2ffa7071944c00a0642fdb9ff77d9cff0da

surements. Performance metrics were collected using PAPI library version 7.0.0[4] [6]. The performance for each micro-benchmark is measured with varied **input sparsity**, the percentage of loop iterations in which the control-flow condition evaluates to true. Results are reported for input sparsity of 2%, 20%, 40%, and 80%, with true predicates randomly distributed throughout the loop iterations. These percentages represent use cases varying from very few active lanes (2%) to mostly active lanes (80%). A random distribution of true predicates means that none of the evaluated approaches can make assumptions about the order or the pattern of taken/not-taken paths. To Apply ALC, input source code is first compiled to LLVM IR using Clang. The produced IR is fed to the transformation pass and all analyses and transformations are done at this level. Then clean up and further optimization passes in the `O3` pipeline are executed to ensure that the transformed IR is fully optimized. Finally, the generated binary file is produced.

The results presented next contrast the performance of: (i) `ALC`, a compiler-generated version of Wyatt *et al.*'s ALC; (ii) `ALC+DP`, the improved version of `ALC` proposed in this work which employs data permutation to eliminate gather instructions; (iii) `if-conv`, `if`-conversion performed by Arm's Clang compiler. A comparison of the `if`-conversion code generated by three production-ready compilers — Arm's Clang, GCC, and Clang — revealed that Arm's Clang generates slightly faster code for the evaluated micro-benchmarks.

### 0.5.3 Data Permutation: More Instructions But Better Performance

This experiment aims to answer ① by comparing the performance of each version of the `if-then-else` micro-benchmark generated by Clang. Figure **??** shows the speedup over `if-conv`. As discussed in Section 0.3.2, gather instructions have higher latency than regular vector loads, thus hurting `ALC`'s performance in comparison to `if-conv`, which only uses regular vector load instructions. Such overhead is quantified in Figure **??**, which indicates that the ratio of stalls due to pending memory operations (w.r.t `if-conv`) is significantly higher

[4] c415d2f1190027c961c904a4305b3eeaacce3df2

in `ALC` than in `ALC+DP`, which explains `ALC+DP` better performance. Also, the address calculations of gather/scatter instructions are executed in the vector units and thus compete for resources and stall other arithmetic operations in `if-then-else`. As Figure **??** indicates, `ALC` has more stalls waiting for resources than the baseline `if-conv` while `ALC+DP` has fewer resource stalls than `ALC` because gather instructions are eliminated via data permutation. In addition, `ALC+DP` outperforms `if-conv` for all input sparsity cases and, in particular, by more than 7% in the 80% sparsity case — a positive answer to question ①. `ALC+DP` performs better than `if-conv` even though it executes more instructions to perform data permutation, as Figure **??** indicates. These data-permutation instructions are vector-to-vector instructions that incur much lower latency than gather instructions do. `ALC+DP` still suffers more memory stalls than the baseline `if-conv` because of the high latency of scatter stores that, usually, cannot be avoided. In contrast, `ALC` always performs worse than `if-conv` and shows performance degradation of up to 6% in the 40% sparsity case.

### 0.5.4 Scatter Instructions Significantly Impact Performance

In this experiment, the code of the `if-then-else` micro-benchmark is modified to have fewer stores — only one per CFDP—, but it has the same number of arithmetic and load operations. Fewer store operations translate to fewer scatter instructions in the compiler-generated ALC code. Therefore, an increase in ALC's performance with fewer scatter operations indicates a positive answer to ②. As Figure **??** indicates, with fewer scatter stores both `ALC` and `ALC+DP` outperform the baseline `if-conv`. Moreover, `ALC+DP` is four times faster than `ALC`, outperforming `if-conv` by up to 79% in the 2% sparsity case, a strong indicator of the effectiveness of eliminating both predicated instructions and gather loads. Figure **??** shows a significant reduction in the number of resource stalls for `ALC+DP` as a direct result of having fewer scatter store instructions and thus explains the speedup over `if-conv`. Stalls due to memory operations follow the same trend as in the `if-then-else` micro-benchmark, as indicated in Figure **??**, evidence that arithmetic operations can amortize the effects of memory stalls. Still, address calculations for gather/scatter instructions compete for resources resulting in a significant number of resource-busy stalls.

### 0.5.5 Data Permutation Improves ALC

This experiment evaluates the ALC algorithm modification discussed in Section 0.3.4 on the `if-then` micro-benchmark. Figure **??** presents speedups of `ALC` and `ALC+DP` over `if-conv`. In this case, both ALC versions are unable to provide improvements over `if-conv` and result in performance degradation. The single control-flow data path case is challenging for any variant of ALC because, unlike the `if-then-else` case that has two CFDPs, considerably fewer instructions are executed with inactive lanes. The results indicate a negative answer to ③, *i.e.*, in general, it is not beneficial to apply ALC on loops with a single CFDP. Nevertheless, `ALC+DP` provides significant speedup over `ALC` by eliminating gather load instructions. Moreover, and similar to the two CFDP cases, both memory and resource stalls are lower with `ALC+DP` than with `ALC`, as Figure **??** and Figure **??** indicate. As seen in Figure **??**, even in a modified version of `if-then` with fewer store instructions, neither version of ALC can outperform `if-conv`. The exception is the very sparse case (2%), where `ALC+DP` provides a speedup of 7% over `if-conv`. Thus, the results indicate that `ALC+DP` can benefit loops with single CFDP and very few true predicates. `if-then`, which has fewer stores, has similar behavior in terms of resource and memory stalls, as Figure **??** and Figure **??** indicate, as the same benchmark with more store instructions.

### 0.5.6 Conditionally Incremented Array Indexes

The ALC analysis pass found many loops in existing benchmarks (e.g. SPEC CPU 2017 [1] and MiBench [4]) that could be legally transformed by ALC transformation. However, the cost/benefit analysis indicated that those loops would not benefit from ALC because they contained not enough instructions to amortize the cost of index and data permutation (see Section 0.4.1).

Another challenge discovered when trying to apply ALC to existing benchmarks that contain loops with conditional statements, such as the ones in MiBench [4], is related to efficient mappings of conditional computations to vector code.

```
1 for (int i = 0, j = 0; i < n; i++) {
2     if (cond[i]) {
3         b[i] = a[j];
4         j++;
5     }
6 }
```

Listing 3: Conditional increment of array indexing variable.

For example, Listing 3 shows a simplified example of a loop pattern that is widely found in the `jpeg` benchmark in MiBench. The loop index into array `a` with variable `j`, which is conditionally incremented when (Line 4) `cond[i]` is true (Line 2). To the best of our knowledge, there is no single instruction in modern vector ISAs that can compute an index vector with the values of `j` in this case. Therefore multiple instructions would be required, which might make it unprofitable to vectorize such loops. Neither Clang, GCC, nor Arm's Clang vectorized the loops in SPEC CPU 2017 with the pattern shown in Listing 3. Because of such non-trivially vectorizable pattern, this work did not find opportunities to apply ALC as a compiler-enabled transformation to these benchmark suites. If future versions of vector ISAs include instructions to create conditionally-strided index vectors then more loops could become candidates for `if`-conversion and ALC. A potential workaround would be to compute the values of `j` on a separate loop, storing such values in a temporary array, and then using the computed array to index `a`. However, hoisting the computation might not always be possible because of intra-iteration dependencies.

## 0.6 Related Work

The seminal work by Allen *et al.* introduced the idea of converting control-flow dependencies into data dependencies [2]. Allen *et al.*'s technique, commonly know as control-flow linearization or `if`-conversion [13], was introduced in the context of parallelizing FORTRAN compilers. Such compilers excelled in exploiting data-parallel loops by identifying data dependencies. Although `if`-conversion can increase the opportunities for vectorization [7], it can significantly waste computational resources. `if`-converted code keeps units busy with computations on vectors with inactive lanes, which correspond to non-taken paths in the original program's CFG. As the results in this work show, ALC outperforms `if`-converted code because it maximizes SIMD utilization by constructing uniform vector and avoiding the execution of linearized code.

Branch-on-superword-condition-codes (BOSCCs) is a common approach that avoids executing vector instructions with inactive lanes [16]. Originally introduced for multimedia extensions, BOSCCs are instructions, or sequences of instructions, that guard the execution of uniform paths. Such paths only contain instructions that would be executed when all lanes are active (or inactive) with respect to the guard condition. BOSCCs can improve on `if`-converted code [17], however, BOSCCs degenerate into `if`-converted code when uniform vectors are infrequent [14]. ALC overcomes this by actively merging non-uniform vector until a unform vector is formed, thus effectively avoiding `if`-converted code. Moreover, BOSCCs can lead to code explosion on loops with many CFDPs. Moll *et al.*'s work addresses code explosion by selectively using BOSCCs and `if`-conversion [12]. Nevertheless, Moll *et al.*'s solution still degenerate into `if`-converted code when uniformity is infrequent or not contiguous (w.r.t. loop iterations).

Wyatt *et al.* made a step forward by actively constructing uniform vectors instead of expecting dynamic uniformity [14]. ALC makes use of SVE instructions to merge active lanes and execute uniform paths. However, in Wyatt *et al.*'s seminal work, there was no in-silicon implementation of SVE, thus all the evaluation was conducted on Arm's instruction emulator (ArmIE). As a result, only decrease in number of dynamic instructions was reported on Wyatt *et al.*'s paper. In contrast, this work evaluates ALC on real hardware with SVE. Besides identifying a major problem in its original design (Section 0.3.2), this work re-designs ALC as a compiler-enabled transformation. Moreover, experimental results show that avoiding gather instructions via data permutation improves on ALC prior design up to 4× (Section 0.5.3). Results also indicate that factors used in the proposed cost/benefit analysis (Section 0.4.1) directly impact ALC's performance. Finally, the new ALC design outperforms `if`-converted code produced by state-of-the-art compilers. Despite these significant advancements, there are still challenges that can limit ALC effectiveness as a compiler pass (Section 0.5.6).

## 0.7 Conclusion

This work presents `ALC+DP`, a redesign of ALC as a compiler-enabled transformation. An in-depth experimental evaluation of ALC on SVE hardware reveals a key design issue of the original ALC design: a high number of memory and resource-busy stalls is caused by gather instructions. `ALC+DP` is a redesign of `ALC` that eliminates gather instructions through the combination of regular vector loads and data permutation. This work contributes an implementation of ALC in the production-ready LLVM compiler framework that includes a cost/benefit analysis to decide *when* and *how* to apply ALC. This analysis considers the number of instructions on each control-divergent path, the ratio of compute and memory operations, and the complexity of the loops CFG, which are key factors that have been shown to impact ALC's effectiveness and efficiency. Experimental results indicate that `ALC+DP` outperforms ALC's previous design by up to $3x$. `ALC+DP` also outperforms `if`-converted code produced by state-of-the-art compilers such as Arm's Clang by up to 79%. Although ALC is implemented in LLVM, its re-design can be integrated into any modern compiler to automatically increase SIMD utilization of `if`-converted & vectorized loops.

,,,,,,,,,,

,,,,,,,,,,,,,,,,,,,,

# References

[1] T. S. P. E. C. (SPEC), *The SPEC2017 Benchmark Suite*, version 1.1.0, Sep. 21, 2019. [Online]. Available: `https://www.spec.org/cpu2017/`.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '83, New York, NY, USA: Association for Computing Machinery, Jan. 1983, pp. 177–189, ISBN: 978-0-89791-090-3. DOI: `10.1145/567067.567085`. [Online]. Available: `https://dl.acm.org/doi/10.1145/567067.567085` (visited on 04/17/2023).

[3] Fujitsu, *A64FX: Microarchitecture Manual*, English, version Version 1.3, Fujitsu Limited, Oct. 2020, 136 pp.

[4] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. DOI: `10.1109/WWC.2001.990739`.

[5] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 1–6, Jan. 1997, ISSN: 0164-0925. DOI: `10.1145/239912.239913`. [Online]. Available: `https://doi.org/10.1145/239912.239913`.

[6] Innovative Computing Laboratory (ICL), *PAPI: The Performance Application Programming Interface*, version 7.0.0, Nov. 14, 2022. [Online]. Available: `https://icl.utk.edu/papi/`.

[7] Jaewook Shin, M. Hall, and J. Chame, "Superword-Level Parallelism in the Presence of Control Flow," en, in *International Symposium on Code Generation and Optimization*, San Jose, CA, USA: IEEE, 2005, pp. 165–175, ISBN: 978-0-7695-2298-2. DOI: `10.1109/CGO.2005.33`. [Online]. Available: `http://ieeexplore.ieee.org/document/1402086/` (visited on 04/14/2023).

[8] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91, Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 93–103,

ISBN: 0897914198. DOI: 10.1145/99583.99599. [Online]. Available: https://doi.org/10.1145/99583.99599.

[9] D. Levine, D. Callahan, and J. Dongarra, "A comparative study of automatic vectorizing compilers," *Parallel Computing*, vol. 17, no. 10-11, pp. 1223–1244, 1991.

[10] B. Liu, A. Laird, W. H. Tsang, B. Mahjour, and M. M. Dehnavi, "Combining Run-Time Checks and Compile-Time Analysis to Improve Control Flow Auto-Vectorization," en, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Chicago Illinois: ACM, Oct. 2022, pp. 439–450, ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569663. [Online]. Available: https://dl.acm.org/doi/10.1145/3559009.3569663 (visited on 04/14/2023).

[11] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua, *et al.*, "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2011, pp. 372–382.

[12] S. Moll and S. Hack, "Partial control-flow linearization," en, in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia PA USA: ACM, Jun. 2018, pp. 543–556, ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192413. [Online]. Available: https://dl.acm.org/doi/10.1145/3192366.3192413 (visited on 04/14/2023).

[13] J. C. Park and M. Schlansker, *On predicated execution.* Hewlett-Packard Laboratories Palo Alto, California, 1991.

[14] W. Praharenka, D. Pankratz, J. P. L. De Carvalho, E. Amiri, and J. N. Amaral, "Vectorizing divergent control flow with active-lane consolidation on long-vector architectures," en, *The Journal of Supercomputing*, vol. 78, no. 10, pp. 12 553–12 588, Jul. 2022, ISSN: 1573-0484. DOI: 10.1007/s11227-022-04359-w. [Online]. Available: https://doi.org/10.1007/s11227-022-04359-w (visited on 04/17/2023).

[15] R. G. Scarborough and H. G. Kolsky, "A vectorizing fortran compiler," *IBM Journal of Research and Development*, vol. 30, no. 2, pp. 163–171, 1986.

[16] J. Shin, "Introducing Control Flow into Vectorized Code," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, ISSN: 1089-795X, Sep. 2007, pp. 280–291. DOI: 10.1109/PACT.2007.4336219.

[17] J. Shin, M. W. Hall, and J. Chame, "Evaluating compiler technology for control-flow optimizations for multimedia extension architectures," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 235–243, 2009, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2009.02.

002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933109000180.

[18] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, pp. 363–400, 2000.